Grundlagen der Informatik III

Wintersemester 2013/2014

Prof. Dr.-Ing. Michael Goesele,
Simon Fuhrmann, Fabian Langguth

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Praktikum

31.10.2013

Das Praktikum wird in Teams von drei Personen bearbeitet, die alle der selben Übungsgruppe angehören müssen. Vermerken Sie im Quellcode alle Teammitglieder mit Matrikelnummern. Die Lösung muss bis **spätestens 19.12.** im Moodle hochgeladen werden. Es genügt, wenn *ein* Teammitglied die Lösung abgibt. Den eigentlichen Testattermin vereinbaren Sie selbst mit Ihrem Tutor.

Geben Sie alle zum Kompilieren und Ausführen Ihres Programms benötigten Dateien in einem .zip- oder .tar.gz-Archiv ab. Ihre Abgabe muss mit den auf dem Aufgabenblatt angegebenen Anweisungen auf den RBG-Poolrechnern kompilier- und ausführbar sein, ansonsten wird sie mit 0 Punkten bewertet.

Wenn Sie dieses Praktikum erfolgreich bearbeiten und die Klausur bestehen, erhalten Sie **1 Punkt**.

Motivation

In graphischen Problemstellungen wie beispielsweise Rendering (z.B. mittels Rasterisierung, Ray Tracing oder Path Tracing) aber auch in Simulationen ist es nötig, Gegenstände der realen Welt im Computer zu modellieren. Die Gegenstände werden dazu in möglichst einfache Bestandteile zerlegt und deren Form durch graphische Primitive wie Punkte und Dreiecke angenähert.

Da eine modellierte Szene in realistischen Anwendungszenarien sehr groß werden kann, die auszuführenden Aufgaben aber in jedem Fall (insbesondere im Echtzeitrendering) sehr schnell durchgeführt werden müssen, ist es außerdem nötig, dass die geometrischen Primitive in einer Datenstruktur abgelegt werden, die effiziente Operationen, wie z.B. Schnitttests, ermöglicht. Aus GdI2 sind Ihnen möglicherweise Datenstrukturen für die schnelle Auffindbarkeit von Daten, wie B-Bäume, AVL-Bäume oder Rot-Schwarz-Bäume, bekannt. Entsprechende Datenstrukturen gibt es auch für 2- und 3-dimensionale Daten im Raum, sogenannte *räumliche Datenstrukturen*. Einige Beispiele sind Octrees, kD-Trees, Bounding Volume Hierarchies, uvm...

In diesem Praktikum sollen Sie eine einfache geometrische Datenstruktur implementieren. Hierbei soll die Verwendung von Klassen in C++ geübt werden.

Überblick

Ihre Aufgabe ist es, eine *Bounding Volume Hierarchy* (BVH) zu implementieren. Diese BVH wird im nächsten Praktikum weitere Anwendung in einem Raytracer finden. Raytracing ist ein Verfahren zur Visualisierung, dem *Rendering*, von dreidimensionalen Objekten im Raum. Hierfür werden Sichtstrahlen von der Position der Kamera (auch *Augpunkt* genannt) durch die Pixel der Bildebene in die Szene gesendet und der nächste Schnittpunkt mit der Szene gesucht. Daher muss die von Ihnen implementierte BVH neben der eigentlichen Erstellung auch Fragen der Form „Welches Dreieck wird vom Strahl r als erstes getroffen?“ beantworten können. Eine ausführlichere Beschreibung zu Raytracing finden Sie im Internet und natürlich im nächsten Praktikum.

Wie schon in den vorherigen Praktika haben wir Ihnen ein Codegerüst erstellt, das Sie nun erweitern sollen. Im Folgenden wird ein kurzer Überblick über den Code gegeben werden. Details entnehmen Sie bitte direkt dem Codegerüst.

Geometrische Primitive und Schnittests

Das von uns erstellte Codegerüst enthält bereits drei geometrische Primitive:

- Dreieck (Klasse `Triangle`): Ein Dreieck wird durch seine Eckpunkte beschrieben. Zur Oberflächenvisualisierung findet die Oberflächennormale (ein 3-Vektor, welcher sekrecht zur Oberfläche steht) Anwendung. Eine Normale kann mit dem Kreuzprodukt aus zwei Dreieckskanten berechnet werden, und wird normalisiert angegeben.
- Axis-Aligned Bounding Box (Klasse `AABB`): Eine AABB ist ein Quader, dessen Kanten mit den Hauptachsen des Koordinatensystems ausgerichtet sind. Deswegen lässt sich die AABB einfach über zwei diagonal gegenüberliegende Eckpunkte a^{min} und a^{max} beschreiben, für die gilt: $a_x^{min} \leqslant a_x^{max}$, $a_y^{min} \leqslant a_y^{max}$, $a_z^{min} \leqslant a_z^{max}$.
- Strahl (Struct `Ray`): Ein Strahl ist eine Halbgerade, die durch einen Anfangspunkt und einen Richtungsvektor, in dessen Richtung der Strahl verläuft, beschrieben wird.

Zudem existiert bereits eine Template-Klasse `Vec3`, die arithmetische Operationen (Addition, Skalar-Prokukt, Kreuzprodukt, etc..) bereitstellt.

Wie sich im Folgenden zeigen wird, werden Schnitttests zwischen den Primitiven Strahl und Dreieck sowie Strahl und AABB benötigt. Diese Tests werden von den `intersect`-Methoden der jeweiligen Klassen implementiert, auf die Sie gerne zurückgreifen können. Sie können die Schnitttests natürlich auch selbst implementieren – achten Sie hierbei aber darauf, dass die Funktionssignaturen, sowie die Funktionalität nicht verändert werden dürfen.

Dateiformat für Dreiecksnetze

Die Dreiecke, die Sie in die BVH einsortieren sollen, liegen in einem *.off* Dateiformat vor. Die Struktur der Datei ist wie folgt:

```
1   OFF
2   4 4 0
3   0.0         0.0         2.0
4   1.632993   -0.942809   -0.666667
5   0.000000    1.885618   -0.666667
6   -1.632993   -0.942809   -0.666667
7   3 1 0 3
8   3 2 0 1
9   3 3 0 2
10  3 3 2 1
```

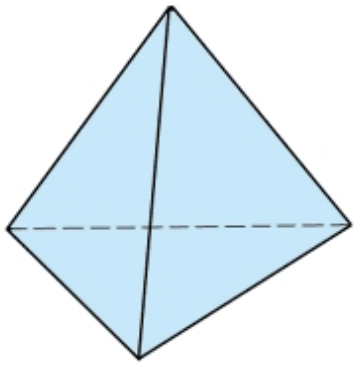

Abbildung 1: Tetraeder

Die erste `4` in Zeile 2 gibt an, dass die Szene aus 4 Eckpunkten (*Vertices*) besteht (jeweils gegeben durch seine 3 Koodinaten, Zeilen 3-6). Diese Eckpunkte haben einen implizieten Index, der Eckpunkt in Zeile 3 hat die Nummer 0, der Eckpunkt in Zeile 4 die 1, etc. Die Dreiecke bestehen aus drei Knoten (daher die vorangestellte `3` in Zeile 7-10), die durch ihren Index angegeben sind. Diese Szene besteht aus insgesamt 4 Dreiecken (Zeile 2, die zweite `4`).

Dadurch kann ein Netz repräsentiert werden, das die Form eines 3D Objekts modelliert. Die Beispieldatei erzeugt einen Tetraeder (vgl. Abbildung 1).

Das Einlesen der Datei übernimmt die Funktion `load_off_mesh(...)` für Sie. Sie speichert die Eckpunkte (3D Vektoren) sowie die Indizes, die die Dreiecke definieren, hintereinander in jeweils einer Liste ab und speichert sie im Struct `Mesh`. Ihr Pendant `save_off_mesh(...)` speichert das übergebene Mesh in einer Textdatei.

Räumliche Datenstruktur

Eine Datenstruktur, in die man geometrische Primitive ablegt, muss sehr viele Anfragen der Form “Welche Objekte befinden sich in diesem und jenem Bereich des Raums?”, “Wenn ein Strahl hier durch den Raum wandert, welche Objekte trifft er dann?”, etc. beantworten. Lineare Datenstrukturen sind hier nicht geeignet, da sie viel Zeit benötigen, um derartige Fragen zu beantworten.

Eine verbreitete räumliche Datenstruktur ist eine *Bounding Volume Hierarchie* (BVH). Ein *Bounding Volume* (BV) ist ein Körper, der eine Menge von geometrischen Objekten umschließt. Entscheidend hierbei ist, dass das BV eine einfachere geometrische Form als die zu umschließenden Objekte darstellt, damit Tests (z.B. Schnitttests) schneller vonstatten gehen. Hierfür bieten sich beispielsweise Axis-Aligned Bounding Boxes (AABB) oder Oriented Bounding Boxes an. In diesem Praktikum betrachten wir nur AABBs.

Die gesamte Szene ist in einer hierarchischen Baumstruktur organisiert, die aus einer Wurzel, inneren Knoten sowie Blättern besteht. Die Blätter beinhalten das entsprechende geometrische Primitiv und haben keine weiteren Kinder. Innere Knoten sowie die Wurzel beinhalten keine geometrischen Primitive, sondern Referenzen auf ihre Kinder. Alle Knoten (inklusive der Blätter) speichern ein Bounding Volume, das die Geometrie des gesamten *Unterbaums* umschließt. Das Bounding Volume der Wurzel umschließt also die komplette Szene. Eine beispielhafte BVH sehen Sie in Abbildung 2.

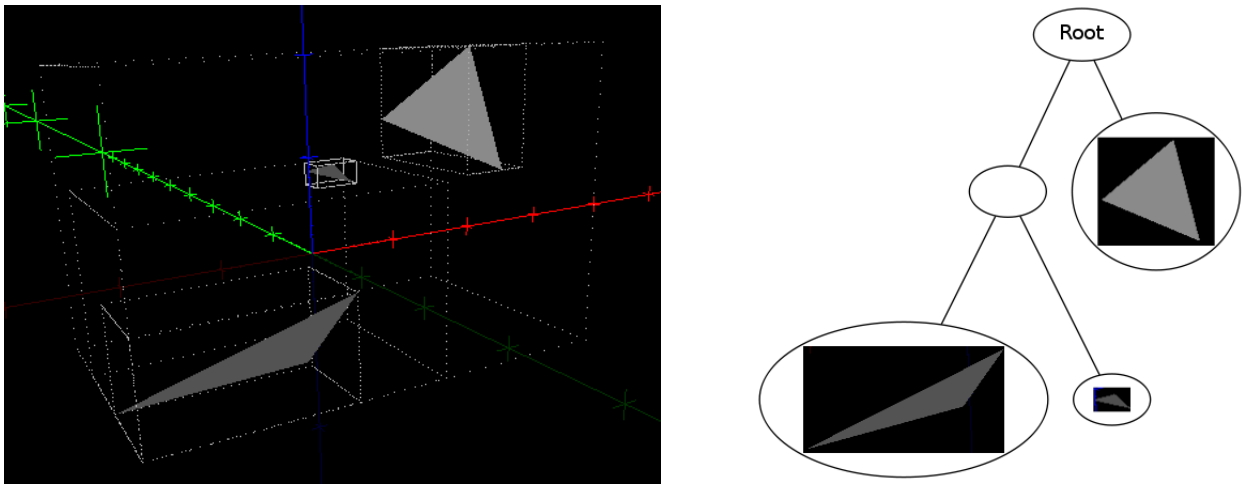

Abbildung 2: Binäre Bounding Volume Hierarchie mit einem Primitiv pro Blatt. Jeder Knoten hält zusätzlich ein Bounding Volume, das die Primitive des kompletten Unterbaum enthält.

Entscheidend für die Performanz der späteren Tests ist die Art und Weise, wie der Baum organisiert ist. Eine *gute* BVH ermöglicht eine schnelle Traversierung und somit eine hohe Beschleunigung der Schnitttests. Überlegen Sie sich, wie viele Dreiecke Sie in jedem Blatt speichern wollen und mit welcher Strategie Sie die Szene aufteilen. Gebräuchliche Strategien sind etwa:

1. Split in der Mitte der längsten Achse: Diese Strategie erstellt in einer Top-Down Vorgehensweise einen Binärbaum. Hierfür findet eine sukzessive Unterteilung der Objekte in je zwei disjunkte Teilgruppen statt. Erstellen Sie eine temporäre AABB, die alle Schwerpunkte der übergebenen Dreiecke enthält. Diese teilen Sie an der längsten Achse und übergeben die Dreiecke der linken/rechten Hälfte an das linke/rechte Kind.

2. Median-Cut: Die Menge der Szenenobjekte wird entlang einer der Koordinatenachsen x, y oder z sortiert und am Median in zwei disjunkte Teilmengen unterteilt, die an die beiden Kinder weitergereicht werden. Die Sortierung der Dreiecke kann zum Beispiel anhand des Schwerpunktes der Dreiecke stattfinden. Für die Wahl der Koordinatenachsen existieren verschiedene Kriterien, zum Beispiel anhand an der längsten Achse oder man wählt einfach alternierende Achsen in jedem Rekursionsschritt. Mehr Details finden Sie zum Beispiel in [KÖ9], Kapitel 3.1.1.

3. Surface Area Heuristik (SAH): Der Schwachpunkt der beiden vorangegangenen Strategien ist die gleichmäßige Unterteilung der Objekte anhand der längsten Achse bzw. des Medians. Die Idee hinter der Surface Area Heuristik ist es, eine Kostenfunktion einzuführen, mit der die Qualität der Objektaufteilung bestimmt werden kann. Die SAH basiert auf der Erkenntnis, dass die Wahrscheinlichkeit, mit der ein Strahl Geometrie trifft, mit der Oberfläche der Geometrie zusammen hängt. Mehr Details finden Sie zum Beispiel in [KÖ9], Kapitel 3.1.2.

Beachten Sie, dass Sie zwei verschiedene Arten von AABBs verwenden müessen. Für die Berechnung der Splits verwenden Sie eine AABB, welche die Zentroide der Dreiecke umfasst, und nicht die Eckpunkte der Dreiecke. Dies stellt sicher, dass bei jedem Split zwei Kinder entstehen, die nicht leer sein können. Im Gegensatz dazu speichern

Sie in der BVH für jeden Knoten die AABB, welche die Dreiecke komplett einschießt, also die Eckpunte der Dreiecke zur Berechnung verwendet. Dies stellt sicher, dass Schnitttests auf der BVH korrekt funktionieren.

Welche Strategie Sie wählen, können Sie selbst entscheiden. Bedenken Sie jedoch, dass die BVH auch noch im nächsten Praktikum relevant ist und dementsprechend performant sein sollte. Wir haben Ihnen zur Hilfe eine Statistikfunktion beigelegt, die Ihnen einige charakteristischen Größen der BVH berechnet und auf der Konsole ausgibt. Zudem können Sie die Zeit für die Erstellung der BVH sowie eines Rendering-Vorgangs messen und so Ihre Implementierung optimieren.

Aufgabenstellung

Implementieren Sie die Klasse `BVHNode`, die ein Mesh entgegennimmt und eine binäre BVH anhand einer Strategie erstellt. Neben der Erstellung der BVH ist Ihre zweite Aufgabe die effiziente Traversierung des Baums, um anhand eines gegebenen Strahls entscheiden zu können, auf welches Dreieck er als erstes trifft. Traversieren Sie die BVH rekursiv beginnend bei der Wurzel. Falls der Strahl ein BV verfehlt, wird eine Teil der Szene nicht getroffen und der Pfad muss nicht weiter verfolgt werden. Andernfalls werden rekursiv die BVs der beiden Kinder getestet. Falls der Strahl auf das BV eines Blatts trifft, so wird der Strahl gegen die beinhaltenden geometrischen Primitive getestet.

Hinweise

Um das Programm zu kompilieren wechseln Sie in einer Konsole in das Verzeichnis, das Ihre Dateien enthält und geben `make` ein. Das `Makefile` übernimmt alles Weitere für Sie.

Um die Korrektheit Ihrer BVH zu testen, haben wir Ihnen die Funktion `addAABBsToMesh` bereitgestellt, die den Baum traversiert und die Kanten der AABBs als Punkte in ein Mesh einfügt. Dieses Mesh können Sie speichern und z.B. mit UMVE[1] oder Mashlab[2] visualisieren. Außerdem können Sie auf die Funktionen aus `visual.cc` zurückgreifen, um Strahlen zu plotten. Beachten Sie, nur kleine Szenen auf diese Weise zu visualisieren! Zusätzlich können Sie die Ausgabe der Statistikfunktion zur (Teil-)Überprüfung nutzen: Besteht die eingelesene Szene aus $b = 100$ Blättern, so gibt es $i = b - 1 = 99$ innere Knoten inklusive Wurzel[3].

Testen Sie Ihre BVH mit der beigelegten Testszene und untersuchen Sie ihre Korrektheit. Sie können sich auch eigene einfache Testszenen konstruieren und diese testen. Die Funktion `renderImage` rendert das übergebene Mesh und speichert die entstehende Grafik in einer .pgm Datei.

Bewertung

Sie erhalten einen Punkt für die korrekte Implementierung der Klasse `BVHNode`. Ihre Klasse muss die Konstante `MAX_LEAF_TRIANGLES` (in `bvh_node.h`) beachten, um die maximale Anzahl der Dreiecke in einem Blatt zur Compilezeit zu definieren.

Literatur

[KÖ9] Dipl.-Ing. Thomas Köhler. Realtime raytracing: Bounding volume hierarchies on graphics cards, Juni 2009. Fakultät für Informatik und Mathematik, Hochschule München.

[1] `http://www.gris.informatik.tu-darmstadt.de/projects/multiview-environment/`, Reiter *Scene inspect*
[2] `http://meshlab.sourceforge.net/`
[3] vgl. etwa `https://de.wikipedia.org/wiki/Binärbaum`, Abschnitt: „Abzählungen“